Lecture 13: Introduction to Neural Networks

Instructor: Aditya Bhaskara *Scribe: Dietrich Geisler*

CS 5966/6966: Theory of Machine Learning

March 8th, 2017

Abstract

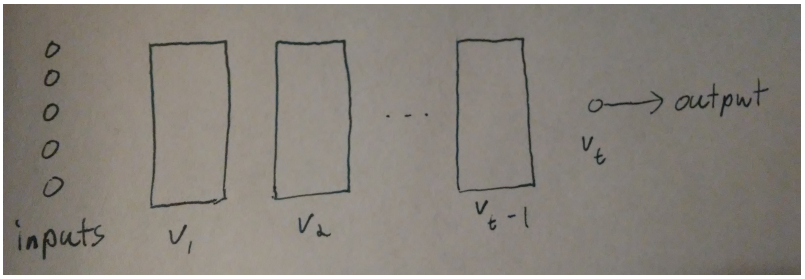
This is a short, two-line summary of the day's lecture. Should provide a rough set of topics covered or questions discussed.

1 Recap: Boosting

In previous lecture, we discussed boosting in learning. Boosting is the principle that, given a weak learner for a class, it is possible to obtain a strong learning. More formally, given a black box of weak learners, it is possible to produce a strong learning algorithm from the collection of weak learning algorithms.

2 Formalizing Feed-Forward Neural Networks

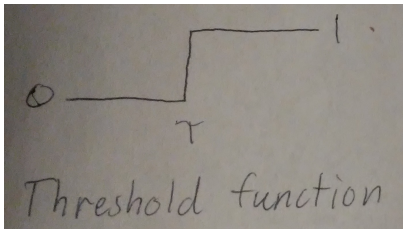
The first type of neural network we will explore is a feed-forward neural network. Feed-forward neural networks take a collection of inputs, pass these inputs through a series of Hidden Layers (HLs) composed of neurons, and use this mapping to produce an output. This output can be of any dimension and does not depend on the dimension of the input.



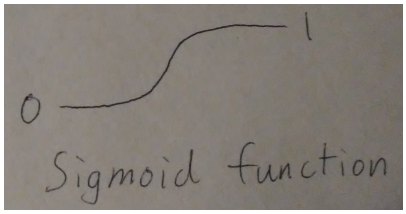
Each neuron in the network is an entity connecting inputs to other neurons (or the output). Each of these essentially functions as a map from a given set of inputs values to an output value. For a given neuron, each incoming input is multiplied by a weight w_i which is modified as new data is given to allow learning. The neuron has some function f by which this value is produced. This means that for some neuron with inputs $x_1, x_2 \ldots, x_k$, weights $w_1, w_2, \ldots, w_k$, and function f, the following output is given:

$$f(w_1x_1 + w_2x_2 + \cdots + w_kx_k) \tag{1}$$

The choice of f arbitrary; however we generally require that $f : \mathbb{R}^n \to [0, 1]$ and expect that f is nonlinear. Common choices for f include the threshold function:



And the sigmoid function:



Other common choices for f exist but will be considered later.

The final output of a given neural network after moving through each layer for some initial inputs $x_1, x_2, \ldots, x_n$ is given by the function, where $f(w_x)$ is a neuron as given in Eq. (1):

(2) $$f_{v_t}(x_1, x_2 \ldots, x_n) = f(w, [f(w, \ldots [f(w, x), \ldots f(w, x)])])$$

This equation implies some useful properties of neural networks. First, since Eq. (2) is deterministic, the result of an arbitrary feed-forward neural network is always deterministic; that is, an unchanged neural network always produces the same value given the same set of inputs. Second, the total function provided by the neural network can be nonlinear, even though each neuron is a nonlinear function of a linear combination of inputs. Finally, the nonlinearity of the resulting function described in Eq. (2) allows for modeling of complicated patterns and interactions.

The power apparently granted by neural networks, however, presumes two necessary questions. First, what does $f_{v_t}(x_1, \ldots, x_n)$ actually look like? Second, can we find a nueral network that computes some f_{v_t} given enough training examples?

3 Structure of $f_{v_t}(x_1, \ldots, x_n)$

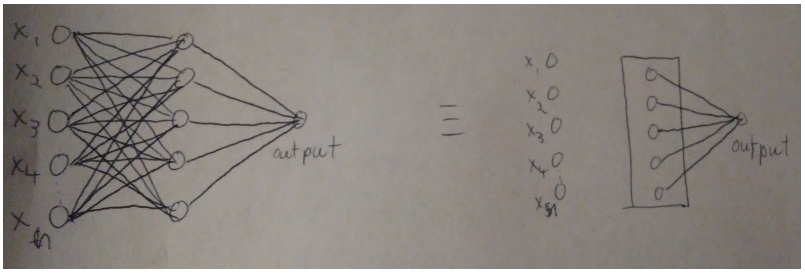
We will denote the number of layers of a neural network with $t \geq 1$ and observe that the number of hidden layers in such a network equal to $t - 1$. Let us first consider a simple neural network, where $t = 1$. This is just a direct mapping from our inputs $x_1, x_2, \ldots, x_n$ to the output. In other words, we just have that $f_{v_t} = f$, where f is the function given in Eq. (1).

We will now examine the neural networks with $t = 2$, noting that this implies we have one hidden layer. Using Eq. (1) and Eq. (2), we obtain the function:

(3)

$$f_{v_t}(x_1, x_2 \ldots, x_n) = f(w_1 f_1(w_{11}x_1 + \cdots + w_{1j}x_j) + \ldots w_i f_i(w_{i1}x_1 + \cdots + w_{ij}x_j))$$

This can be visualized as follows (note that the two diagrams are equivalent):44



The activation function given in Eq. (3) can represent any convex shape as a conjunction of linear classifiers, an illistration of which can be seen below:

More generally, however, [Cybenko '89] and [Kolmogorov '50s] tell us that any arbitrary boolean function can be approximated to arbitrary accuracy by two-layer threshold neural networks. Showing this result is fairly technical, however, so we will instead illustrate a similar case where we assume boolean functions over the domain $\{0,1\}^n$. Specifically, we will show that any boolean function $f : \{0,1\}^n \to \{0,1\}$ can be written as a two-layer threshold network.

The idea of this proof is as follows. Let us suppose that $n = 3$, which gives that inputs are of the form (x_1, x_2, x_3); thus a total of $2^3 = 8$ inputs are possible. Since only outputs in the set $\{0,1\}$ are possible, some number (protentially 0) of these inputs produce a value of 1 and the remainder produce 0. Then we simply construct a neural network where we have a neuron corresponding to each output of 1 such that these neurons are only fired when precisely the input associated with this output is given. We then produce as our output 1 if at least one of these neurons fires and 0 otherwise.

As an illustration, let us suppose that the inputs $(1,0,0)$ and $(0,0,1)$ produce a value of 1 and all other inputs produce a value of 0. We then construct a network with two neurons with the following functions, respectively:

$f_1 = (x_1) + (1 - x_2) + (1 - x_3) \geq 3$

$f_2 = (1 - x_1) + (1 - x_2) + (x_3) \geq 3$

$f_{v_t} = f_1 + f_2 \geq 1$

Where each neuron provides a 1 if the given condition is true and 0 otherwise. Then we have perfectly modeled our input function.

This argument of course does not depend on n, which means that we can perform a similar operation on any size of input layer. We can therefore conclude that, so long as we don't restrict the number of nodes in the hidden layer, we can perfectly model any boolean function with a two-layer neural network.

This begs the question, however, of what happens when we do limit the number of nodes in the hidden layer of our neural network, as we would have to in the real world? As might be expected from our construction above, there are indeed boolean functions that cannot be represented by two-layer neural network when we limit the number of neurons the hidden layer. A formalization of this idea is given in the theorem below.

3.1 THEOREM. *There exists a boolean function f over n bits to which any threshold network computing f has size $(V + E) \geq 2^{\Omega \ln n}$*

Proof. Before demonstrating this, however, we must bound the VC dimension of our limited networks. Define $smallnets_d = \{$The set of all threshold nets that have $V + E \leq d\}$.

3.2 THEOREM. $VC(smallnets_d) \leq d \log d$

Proof. The proof of this theorem is a bit technical and can be found in the textbook. □

This proof is non-constructive since it is an argument by counting

Now that we have established this bound, we may proceed with demonstrating our original theorem.

We first recall the intuition that the VC-dimension α of a learner gives the number of distinct parameters that can be learned by the algorithm.

(*)

Consider the case where $d \log d < 2^n$ and observe that the alternative where $d \log d \geq 2^n$ trivially shatters all boolean functions with fewer than 2^n inputs.

Then we claim that the class of functions $smallnets_d$ excludes some boolean function with 2^n inputs. But the VC-dimension(H) is the size of the largest set that H can shatter. By (*), VC-dimension$(smallnets_d) < 2^n$. This means that for any set S of inputs of size 2^n, $smallnets_d$ cannot shatter S. □

Before moving to the next section, it is worth summarizing what we have discovered. Simple (small-depth) neural networks initially seem very powerful, and indeed allow representation of a variety of interesting functions. Real-world limitations, however, enforce bounds on the number of neurons and, therefore, the complexity of the functions that can be represented.

We conclude by notin that depth is not sufficient to solve this problem, though it can help. Consider the question: do depth 4 networks (say size n) compute functions that can be computed by size 2 neural networks of size n^2?

4 Learning Power of Neural Networks

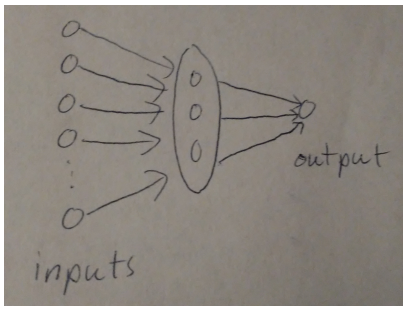
Recall our question from earlier, namely: can we find a nueral network that computes some f_{v_t} given enough training examples? More precisely, is training neural networks a 'hard' problem and, if so, how hard can it be to obtain arbitrary precision?

Suppose we fix a neural network such that $depth = 2$, that is the network has two hidden layers. Can we "train" this network? More precisely, can we find

the optimal neural network to model some function given sample data? This problem is known as Empirical Risk Minization (ERM).

We know the structure of the network, namely that we have two hidden layers and a set number of neurons. Our problem, then, is to find the values for w_i and t_i for all i such that we minimize the number of mistakes across the training set. This problem is generally very difficult, even for fairly simple networks such as the one given!

4.1 THEOREM. *'Even' a network with just* 3 *neurons in the hidden layer is NP-Hard to "learn"; that is, we can find input arrangements where finding the optimal neural network with this arrangement is NP-Hard. More generally, the ERM problem is NP-Hard.*



Proof. We will perform a reduction from the 3-coloring problem, a problem that is known to be NP-Complete. The 3-coloring problem is as follows. Given a graph $G(V, E)$, determine if it possible to color $v \in V$ with the colors red, green, and blue such that $(\forall e \in E)(e = (u, v) \Rightarrow u, v$ are colored with different colors). In other words, this coloring partitions the vertices of the graph in such a way that no edge maps between two elements of the same partition.

Now, to show that the ERM problem for a two-layer neural network with 3 neurons is a reduction from the 3-coloring problem, take a graph $G(V, E)$ and determine some training examples. In particular, it is possible to select those training examples such that ERM=0 iff $G(V, E)$ is 3-colorable. As a hint, if a graph is 3-colorable, some edges to the output will be 1 while the rest will be 0. That is, we have two kinds of input $(0, 0, \ldots, 1, \ldots 0, 0) \to 0$, where 1 occurs at the ith position, and $(0, 0, \ldots, 1, \ldots, 1, \ldots 0, 0) \to 1$, with 1s at the locations of the ends of a given edge (u, v). This will give that every incoming edge ahas an outgoing edge matching two the correct color as given by these inputs. □